`jojsh` and related systems

Joseph Helfer

May 14, 2010

Contents

1 Introduction **3**
- 1.1 What is a shell? . . . 3
- 1.2 Linux vs Unix . . . 3

2 Important Unix and C concepts **4**
- 2.1 System calls and the POSIX standard library . . . 4
- 2.2 "Everything is a file" and file descriptors . . . 4
- 2.3 Streams, stdin, stdout, and stderr . . . 5
- 2.4 Terminals . . . 6
- 2.5 `fork()` and `exec()` . . . 6
- 2.6 `pipe()` . . . 7
- 2.7 The command line utilities . . . 8

3 So what does it do? **8**
- 3.1 Included features . . . 9
  - 3.1.1 Running programs . . . 9
  - 3.1.2 Built-in commands . . . 9
  - 3.1.3 Setting the current directory . . . 9
  - 3.1.4 Exiting . . . 10
  - 3.1.5 Redirection . . . 10
  - 3.1.6 Pipelines . . . 11
  - 3.1.7 Programming structures . . . 12
  - 3.1.8 Variables, parameters, and `export` . . . 12
  - 3.1.9 `if` and `while` . . . 14
  - 3.1.10 `&&` and `||` . . . 14
  - 3.1.11 `for in` . . . 15
  - 3.1.12 Command substitution . . . 15
  - 3.1.13 Programmable prompt . . . 15
  - 3.1.14 Miscellaneous features . . . 16
- 3.2 Missing features . . . 16
  - 3.2.1 Command line editing . . . 16
  - 3.2.2 Completion . . . 17
  - 3.2.3 Command line history . . . 17
  - 3.2.4 Globbing . . . 18
  - 3.2.5 Job Control . . . 18
  - 3.2.6 Other features . . . 18

4 jojsh **19**
- 4.1 Parsing . . . 19
  - 4.1.1 Grammars . . . 19
  - 4.1.2 Parse Trees . . . 21
  - 4.1.3 Words and Pieces . . . 22
  - 4.1.4 The Tokenizer . . . 24
  - 4.1.5 Second-pass parsing . . . 25

4.1.6 The parser functions 27
4.1.7 `parseStmts()` 27
4.1.8 `parseCmd()` 29
4.1.9 `parseWord()` 30
4.1.10 `parseBack()` 31
4.2 Variables 32
4.3 Running 33
4.3.1 `expandWords()` 33
4.3.2 `expandBack()` 34
4.3.3 `runCmd()` 35
4.4 Cleaning 38
4.5 The main loop 38

5 Concluding remarks **39**

A Device Files **40**

B Programmable prompt codes **40**

C Hash tables **41**

1 Introduction

1.1 What is a shell?

For my independent study project, I have written a Unix shell in the programming language C. This is because a shell is one of the most important programs in a Unix environment, and writing one displays many of the most important aspects of a Unix environment and of C.

So what is a shell? You can think of the shell as the most basic program which runs on an operating system. It is what the user interacts with. What should such a program be able to do? Well, at the very least, it should allow the user to run other programs. With this, you can do whatever all the programs on your computer can do; without, it, you can't do anything.

Traditionally, before graphical user interfaces were common, and for some time afterwards, the way in which the user interacted with the operating system was through a command line interface (often abbreviated CLI). It is this type of command line program which I refer to as *shell*, and it is this type of program which I have written. On modern operating systems such as Windows, Mac OSX, and Linux, the shell has been superseded by graphical user interfaces as the primary means of interacting with the computer. The shell is generally used, if at all, inside a window within the graphical interface. On certain systems such as Windows, it is then used to accomplish certain tasks which can't be accomplished through the graphical interface, and also as a fallback in case the graphical interface should for some reason fail.

On Linux and similar systems, however, the shell is still quite popular as a primary means of getting things done, as it is considered a better way of accomplishing certain tasks (though whether and why this is the case is a discussion outside of the scope of this paper). Furthermore, on such systems, there is a clearer distinction than in Windows between the operating system and the graphical interface, and for example, a fully functional and capable system can be run in Linux without a graphical interface.

1.2 Linux vs Unix

I have referred to the shell I have written as a "Unix shell". I have also made reference to the operating system Linux. Some clarification is needed here as to the nature and significance of these things. Unix is an operating system created in the 70s. It has a rich and complex history (which, interesting as it is, is outside the scope of this paper), which has resulted in the existence of many "Unix-like" systems – that is, operating systems that share the core Unix structure and functionality – some of which are free and others which are proprietary. By far the most popular free personal Unix-like system is Linux. This is the operating system I use, and it is the one in which I wrote this shell.

Over time, shells on Unix developed in such a way that today a "Unix shell" is clearly recognizable as such and has a certain set of fairly standard features. So even though the shell which I have written is technically a "Linux shell"

in that it was written and runs on a Linux system, the fact of the particular Unix-like system on which I have written this shell has very little bearing on the program itself, and so I call it a "Unix shell" because it is in the style of these classic and standard Unix shells (also because it sounds cool). I should note that I may refer to the system in which this shell is running as Linux or Unix interchangeably.

2 Important Unix and C concepts

Here I will outline some important aspects of the Unix system and C which are relevant to the shell and essential to understanding its workings. I am, however, assuming an understanding of general concepts of programming and of the C language, including functions, flow control, pointers, memory allocation (`malloc()` and `free()`), structures, header files, macros, etc.

2.1 System calls and the POSIX standard library

There is a set of C functions included in every Unix-like system called system calls (or syscalls). These provide a means by which a program can perform low-level functions, for things like opening files and creating processes.

These system calls are a part of an extension to the C standard library called the "POSIX standard library" ("POSIX" being the name of a standard describing Unix-like systems). Besides the Unix system calls, this library provides a multitude of other useful system-related tools, for such tasks as manipulating environment variables, obtaining information about users etc. It is possible that if I refer to the "standard library" below, I really mean the POSIX library combined with the standard standard library.

2.2 "Everything is a file" and file descriptors

In Unix there is an important concept which is that "everything is a file". This means that generally any external thing with which a program interacts and exchanges data will be treated by the program exactly like a file. So how do programs treat files?

At the most basic level, programs interact with files primarily with four system calls (§2.1): `open()`, `close()`, `read()`, and `write()`. The `open()` system call takes as an argument the name of a file on the file system, and returns an integer called a *file descriptor*. Every process has a table which associates every open file with a file descriptor. The program itself doesn't have access to this table. All it knows is that if it opens a certain file, the file descriptor returned will be associated with that file until it is closed. So, for example, the function `close()` simply takes an integer, which is the file descriptor of some file which has been opened. The `read()` and `write()` system calls take an integer, and either take a string to be written to, or return a string read from the file whose file descriptor is that integer.

Because "everything is a file", this exact same process will be used in, for example, interfacing with devices. The device is represented by a file on the file system. This file is opened with `open()`, then to send data to the device (producing sound in a speaker, for instance), I `write()` appropriate data to the corresponding file descriptor.

Another important system call related to file descriptors is `dup()`. `dup()` takes a file descriptor, and "duplicates" it, so that there are now two file descriptors representing the same file. We will see the significance of this later on.

2.3 Streams, stdin, stdout, and stderr

Most of the reading and writing in everyday C programming is not done with low-level system calls and file descriptors, but with streams. Streams are a system-independent input/output mechanism which is part of the C standard library and which serves to create a consistent interface across systems, making the nitty-gritty details of I/O transparent to the programmer. I won't get too much into the details of streams. Suffice it to say that a stream is opened with the function `fopen()`, which itself calls `open()`, gets a file descriptor (which the caller of `fopen()` doesn't see) and returns a special stream structure. It is on these structures which the common output functions like `fprintf()` (for printing formatted text on a specified stream) operate.

More common than `fprintf()` is the function `printf()` which doesn't take a stream as an argument, but just prints the specified text "to the screen" (exactly what this means will be made clear soon). The way that this is accomplished is with two global stream variables, defined in the header file `stdio.h`, and called `stdin` and `stdout` (which stand for standard input and standard output). A call like `printf("string")` is then equivalent to a call of `fprintf(stdout, "string")`. Similarly, the function `getchar()`, which gets a character from standard input, is equivalent to the more general `getc()`, which can operate on any stream.

On a Unix system, the `stdin` and `stdout` streams are guaranteed to be associated, by default, with the file descriptors 0 and 1, respectively (we will soon see the significance of this as well). So file descriptor 1 is associated with some "magical" file which prints to the screen whenever it is written to, and file descriptor 0 is associated with some "magical" file which returns the user's input every time it is read from. The nature of this magic will be elucidated in the next section.

There is also a third stream defined by default in `stdio.h` called `stderr`, which is associated with file descriptor 2, and used mainly to print error messages. Printing to this file descriptor usually just puts the output in the same place as `stdout`, but it is nonetheless useful for various reasons, one of which will be discussed in §3.1.5.

2.4 Terminals

All this standard input and output stuff seems a bit strange and esoteric. It is rare, however, that some aspect of the Unix system isn't totally transparent, and this is no exception.

The shell is a text-based program. The user interacts with it via a *terminal*. Traditionally, a terminal was a physical machine which connected to the computer, and which had a keyboard and a screen capable of displaying characters. Before that, teletypes were used, which had a keyboard for input and a typewriter printout for output. This goes to show that it truly is *text* based, and there are no graphical components to the shell at all. Today, we don't use terminals.

In Linux, the shell interacts instead with a *Virtual Terminal*. This is generally a window inside the graphical windowing system which displays characters just like a real terminal or a teletype. The virtual terminal presents to the program an interface which is indistinguishable from that of a real terminal. Therefore, the type of terminal is completely transparent to the program. This interface is simply a special file on the file system which represents the terminal. Traditionally, then, this would be a device file which, when written to, would send the written data out to the terminal and, when read from, would read data sent in from the terminal. Now, of course, rather than sending and receiving data from a real device, this file sends it to and receives it from the virtual terminal, which is just another process.

So to say that `stdout` "goes to the screen" is really just to say that the open file associated with file descriptor 1 is the special file representing the virtual terminal which the shell is connected to. The same goes for standard input*.

2.5 `fork()` and `exec()`

The system calls `fork()` and `exec()` are what effect process creation in Unix. This is obviously important to our discussion, since the primary purpose of the shell is to launch other programs.

The only way to create a new process in Unix is with `fork()`. `fork()` creates a new process which is an exact replica of the calling process, with a few slight differences. The most important difference is the value returned by `fork()`. Because the new, duplicated process (called the *child*), and the parent process are (almost) identical, after `fork()` has done its job, both processes think that they have just called `fork()`, and expect a return value. `fork()` is designed in such a way that to the child, 0 is returned, and to the parent, it returns the child's process ID (a unique number associated with each process). This lets each process determine whether it is the parent or the child, and act accordingly. A call to fork will therefore look something like this:

*It may seem that I haven't actually explained any of the magic, but just lumped it on this amazing "device file". This is a legitimate concern. I give a brief discussion of this in Appendix A, because its too big for this footnote

```
if(fork()) {
    ...
    //parent code
    ...
} else {
    ...
    //child code
    ...
}
```

The purpose of `exec()` is to replace the binary associated with the current process by a different, specified binary. That is, if some program `foo` calls `exec()` with `"/bin/goo"` as an argument, then the program `foo` will be running no more, and instead the program `goo` will run in its place, having the same process ID as `foo` once had. It is with a combination of `fork()` and `exec()`, then, that a new program is launched. The code that accomplishes this would look something like the following:

```
if(fork() == 0) {
    exec("/bin/some-program");
}
// parent code
```

You'll notice that in the above code snippet, there is "parent code" after the call to `fork()`. This is because the child process calls `exec()`, thereby turning into a different program, and so it never gets a chance to execute the "parent code" afterwards.

You might also have noticed that in the first code snippet, the condition for the `if` is `(fork())` rather than `(fork() != 0)`. This is of course just one of those tricky shortcuts that one can make in C, because booleans are defined in C in such a way that a value of 0 represents `false` and a non-zero value represents `true`, so that `(x != 0)` is equivalent to `(x)`.

2.6 `pipe()`

`pipe()` is another important system call. It provides a primary means of inter-process communication. `pipe()` returns two integers (actually, it takes an array of two integers as an argument, and stores the two values therein). The two integers it returns are file descriptors. Each of these file descriptors represent opposite ends of a one-way "pipe" maintained by the operating system, into which data can be written and out of which data can be read. The first one (that is, the one with array subscript `[0]`) is the "read end" of the pipe, and the second is the "write end". Just on its own, `pipe()` does not provide any means of inter-process communication, but rather a means of *intra*-process communication, because a single process has both ends of the pipe. But when combined

with `fork()`, it can serve for communication between processes. This can be accomplished as follows:

```
/* declare the array to hold the file descriptors from pipe() */
int pipefd[2];

pipe(pipefd);
if (fork()) {
  close(pipefd[0]);
  write(pipefd[1], "Hello, child!\n", 13);
  close(pipefd[1]);
} else {
  close(pipefd[1]);
  read(pipefd[0], NULL, 13);
  close(pipefd[0]);
}
```

The second and third arguments of `read()` and `write()` are, respectively, a buffer into or from which the data should be read, and the number of bytes to be read from or written to the specified file descriptors.

The two branches of the `if`, of course, represent two different processes. The parent closes the read end of the pipe, because it no longer needs it, and the child closes the write end. The parent writes to its remaining end of the pipe, and the child reads from its remaining end. Thus inter-process communication is achieved. Afterwards, each process closes the remaining end of the pipe as it no longer needs it.

2.7 The command line utilities

The usefulness and popularity of shells on Unix systems is not just due to the shells themselves. Just as important is the huge set of standard programs that come with any Unix system. These are various small utilities that have been around on Unix for a long time and are a fundamental part of the Unix experience. These include programs like `ls`, which lists the contents of a directory, `cp` and `mv`, which copy and move files, `cat`, which concatenates files, `cc`, the C compiler, `tar`, an archiving tool, `compress`, a compression tool, and many others. It is very much in the spirit of Unix that each of these tools performs a very simple, specific, and clear task, but can be used in conjunction to accomplish larger and more complex tasks (how they can be used in conjunction, we'll soon see, §3.1.6).

3 So what does it do?

I said earlier that all a shell really needs to be able to do is run other programs. But I also said that a shell can be used as a complete replacement of a graphical interface, and mentioned a certain standard set of features of Unix shells.

So what are these features? I will list here some of the most important and common features of shells – first the ones which I have included, then certain ones that are missing. I will give a brief description of each feature here, and the implementation of the ones I have included in my shell will be explained in the following section.

3.1 Included features

3.1.1 Running programs

This is by far the most important and the fundamental role of the shell. To explain how this is accomplished, I should give a brief overview of what the shell is like. When the shell is started, it presents the user with some kind of prompt. It may be as simple as this:

```
$
```

The user types a command at the prompt, and the shell executes it. The content of these commands is generally the name of a program to execute, but as we will see later, there are other commands as well. Any space-separated words typed after the name of the program are called *arguments*, and are passed to the program (by the `char **argv` array in the case of C). These are generally used so as to allow the user to specify certain options to a program. If, for example, a program operates on a certain file, it would take the path to that file as an argument. Besides indicating files, arguments are often "flags", which generally consist of a hyphen followed by a single letter. So `ls -r` ("reverse"), for example, gives me a listing of files in reverse alphabetical order, whereas `ls -l` ("long") gives me a listing with extra information. There are many other kinds of arguments that programs take, and it varies from program to program.

Two or more commands can be entered on the same line, by separating them with semicolons (`;`).

3.1.2 Built-in commands

As well as being able to run any binary available on your system, shells generally also recognize a set of commands which cause the shell itself to perform some task. They are generally implemented in such a way as to make their operation exactly like that of a normal binary. That is, the built-in command is some word, and it can take arguments which determine or alter what it does. Some built-in commands are just replicas of executable that generally already exist on most systems, while some perform tasks that can only be performed by the shell itself. I have included none of the former category, and very few of the latter. Those that I have included are described below.

3.1.3 Setting the current directory

One of the attributes which each process in a Unix system has is what is called its *working directory*. This is just a certain directory in the file system "in

which" the program can be said to be running. The program might ignore this, but it might make use of it too. For example, the program `ls` will list the files in the current working directory if no directory is specified.

Changing the shell's working directory is accomplished by means of a built-in command, traditionally called `cd`. It takes one argument, which is the directory to switch to. I said earlier that all of the built-in commands I included were of the sort that must be performed by the shell itself. It is clear why this is the case for `cd`; if `cd` were an external program, all it would be able to do would be to change the working directory of *that program*, which would then exit. What needs to be done is to switch the working directory of the *current* process (that is, the shell), so the shell itself must do it. Because the user can change the directory, and the shell always has a current working directory, the user gets the feeling of being "in" a directory at any given time (much like in a file browser) and being able to "move around" the file system.

3.1.4 Exiting

Shells must be able to exit, and this is accomplished by a built-in command called (surprisingly) `exit`. This must be a built-in and not an executable for very much the same reason as `cd` (§3.1.3).

3.1.5 Redirection

One of the most basic and useful features of a shell is to be able to redirect the input and output of programs. I said before that the standard input and output of programs are treated just like files (§2.3). It is not hard to believe then, that the file associated with file descriptor 1 (and therefore the file associated with standard output) can be changed from the terminal to some file on the file system. The typical syntax for this in shells is with the character `>`. So, for example, if I want to list the contents of the current directory and store it some file (`foo`), I would issue the following command:

```
$ ls > foo
```

Similarly, `<` is used to use a certain file as the input to a program, instead of getting input from the terminal. `>>` is used to append output to a file (whereas `>` overwrites the file if it already exists).

Redirection is not limited to just file descriptors 0 and 1. By prepending the `<` or `>` by a number, the specified file is opened for reading or writing (respectively) on that file descriptor. Moreover, redirection can be performed from one file descriptor to another one. This is accomplished by appending the `<` or `>` with an ampersand (`&`) and supplying as the file name the desired file descriptor. For example,

```
$ some-program > somefile 2>&1
```

would redirect both standard output *and* standard error into `somefile`, rather than just standard output. From this, we can gather why standard error is

useful, despite the fact that it normally goes to the same place as standard output; if the above *isn't* done, but instead only standard output is redirected, standard error remains on the terminal, and the program can still print error messages for the user to see, even while its regular output goes into a file.

It may seem strange that the redirection of standard output into a file came *before* the redirection of standard error into standard output. This is due to the slightly unintuitive way in which redirections happen. What really happens when we redirect file descriptor a into file descriptor b is that the file associated with file descriptor b is copied into a. So when we redirect file descriptor 2 into 1, the file associated with descriptor 2 is closed, and whatever file is associated with 1 is copied (via `dup()`, §2.2) into 2. Thus, file descriptor 2 will now point to whatever file file descriptor 1 is associated with, creating the apparent effect that file descriptor 2 is being channelled into 1. By default, file descriptors 1 and 2 both point to the same file (the terminal) so redirecting one into the other will have no effect; file descriptor 1 must *first* be associated with `somefile`, and *then* copied into 2. In general, then, one must be careful with the order of redirections.

3.1.6 Pipelines

It might be argued that pipelines (or just "pipes") are the most powerful feature of Unix shells. I mentioned before that the different basic standard Unix utilities can be combined to perform complex tasks. It is with pipelines that this is accomplished. Pipes serve to redirect the output from one program directly into the input of another program. It may not surprise you that this is accomplished with `pipe()` (§2.6).

Just as with shells in general, there is nothing inherently very powerful about pipelines, but they become powerful by virtue of the standard set of Unix tools. Many of these tools just perform text-manipulation. A salient example is `grep`, which takes as an argument a pattern, and prints out each line from the input which matches the pattern. So, for example, if I want information on a specific process I can use `ps` (which lists information about all the processes) and then `grep` for the name of the specific program I'm interested in.

It is not always text that is piped. Sometimes raw data is as well. For example, if I want to archive a bunch of files, and then compress the resulting archive, I can have `tar` (the archiver) send it's output to standard out instead of a file, and then pipe that output into `compress` (the compressor), which I can have take it's input from standard in.

Pipelines are traditionally indicated with the vertical bar, (`|`). Two or more pipelines can, of course, be used in series. As an example, the following command deletes all `mp3` files in the current directories and all sub-directories:

```
$ find | grep -i '\.mp3$' | xargs rm
```

3.1.7 Programming structures

Another one of the most powerful features of shells is *scripting*. Scripting consists of putting a series of shell commands in a file, and then using that file as the input to the shell, so that it executes all the commands in order. Thus the script acts a sort of program.

In order to make these scripts do anything really interesting, however, it is useful to have features which allow dynamic behaviour, like a program. So certain "programming-language-like" features are typically included in a shell. These specific features are described below.

Shell scripts can generally also be run – instead of redirecting them as the input to the shell, which is a bit clumsy – by supplying the name of the script as the first argument to the shell. Unix also provides an even nicer way of running scripts, which is using the so-called "shebang" (which is short for "hash bang" and refers to the two characters `#!`). If a file is run as an executable, and the first two bytes of that file are a shebang, Unix will treat the rest of the first line (that is, all the bytes up to the first newline byte) as the name of an executable, and execute it with the path to the file being run as the sole argument. For example, if I write a file whose first line is `#!/some/program`, then running this file (with `./filename`) will be equivalent to running `/some/program ./filename`. If the named executable happens to be a shell, the shell will treat the file as a script and run it. Therefore, you can start your scripts with a

```
#!/bin/sh
```

and then just run the script from the command line, as if it was a binary in its own right. When the shell runs such a script, this first line has no effect, because it is treated as a comment (see §3.1.14).

A final way to run scripts is by "sourcing" them. This involves using a built-in command (usually called `source` or `.` – that is, the name of the command is the single character "`.`") and providing as an argument the name of the script to be executed. The difference with this method is that it runs the script *in the current shell*, rather than starting a new process so that so that any changes to the shell environment will persist after the script finishes (for example, and variables set in the script (§3.1.8) will still be set when the script finishes).

Regardless of the method used to run the script, any additional arguments given on the command line are passed to the script to be used if desired (except, in the case of my shell, if the method is sourcing, in which case additional arguments are ignored, because I'm lazy).

3.1.8 Variables, parameters, and `export`

The ability to store values inside variables is one of the most basic features of a programming language, and accordingly, shells generally provide this capacity. The syntax for assignment is typically

```
$ varname=value
```

where `value` is any string*. To "expand" a variable (that is, retrieve it's contents), the name of the variable is prefixed with a `$`, and optionally surrounded by curly brackets (`{}`). For example, the following executes the command `"ls -l"`:

```
$ a=ls
$ b=-l
$ $a ${b}
```

Any environment variables that are set at the time the shell is launched are generally set as variables in the shell so that they can be accessed on the command line. So, for example, if I want to use the locations of the default executable search paths in some command I just use `$PATH`.

Shell variables also provide the means to set environment variables in programs that are launched. This is accomplished with the built-in command `export`. `export` takes as an argument the name of a shell variable and, as the name suggests, "exports" it to the environment. So for example, if I issue the commands

```
$ PATH=/bin:/usr/bin
$ some-program
```

I will not have changed the value of the `PATH` environment variable in `some-program`. If, however, I do

```
$ PATH=/bin:/usr/bin
$ export PATH
$ some-program
```

I will have changed it.

Variables are in reality just a special case of so-called *parameters*. Non-variable parameters cannot be assigned to like variables, but are accessed in just the same way as variables. They fall into two categories:

i) Positional parameters

   These parameters are just numbers (that is, they are accessed with `$0`, `$1` etc.). They hold the command line arguments which can be passed to shell scripts, mentioned in the previous section (§3.1.7).

ii) Special parameters

   The names of these parameters are non-alphanumeric, and they hold some kind of useful information. I have included very few of these – in fact, just two: `$`, which holds the process ID of the shell, and `?`, described in the next section (§3.1.9).

*in some shells it is `set varname=value`

3.1.9 `if` and `while`

The typical format of an if statement in Unix shells is

```
$ if CONDITION; then BODY; [else ELSEBODY;] fi
```

`BODY` and `ELSEBODY` consist of any number of semi-colon or newline separated commands. The `CONDITION` is also a command. It may seems strange that a command could represent a condition. This is possible because every executable returns an integer called an *exit status* to the environment upon completion (this is, for example, the use of value returned from the `main` function in C programs). Similar to the way booleans are implemented in C – but reversed – a value of 0 is used to represent `true`, while a non-zero value is `false`.

Accordingly, it is convention for programs to return an exit status of 0 upon "success" (whatever that may mean), and some non-zero value (often 1) upon "failure". As an example, I could use `"ps | grep some-program"` as my condition, if I want some part of a script to run only if `some-program` is currently running, because `grep` only returns success if it successfully finds the indicated pattern.

`if` statements become particularly useful in combination with a standard utility called `test` or `[` (that is, the name of the command is the single character '`[`'). This is a program which can test a variety of conditions, and returns success if the condition is true. These conditions can be certain assertions about files (such as that a file exists, or is a directory). But more importantly, they can be comparisons, such as the equality of two strings, or greater than or less than between two integers. This allows us to use our shell variables. For example the command

```
$ [ $a -gt 6 ]
```

will return success only if the variable `a` happens to hold a number (remember, shell variables are strings) which is larger than 6. This allows us to use variables to control the flow of a shell script, just like in "real programs".

If the exit status of a program is not used as a condition, it is typically stored inside a special shell parameter (§3.1.8) called `?`.

The syntax for a while loop is typically

```
$ while CONDITION; do BODY; done
```

3.1.10 `&&` and `||`

Another way to make use of the exit status of commands is with logical ands and ors. These allow you to run a command only on the condition that some other command exits successfully (in the case of "and") or unsuccessfully (in the case of "or"). This is accomplished by placing a `&&` or `||` between the two commands, the second command being the one contingent on the first's exit status. It may not seem immediately obvious why this has anything to do with logical ands and ors, but with some reflection, we see that this is just the same way in which

short-circuit evaluation works in C and other languages. For example, given some command such as

```
$ command1 && command2 && command3 && command4
```

each command will be evaluated until one exits unsuccessfully, at which point the chain will stop. Since the exit status of the whole sequence is that of the last command executed, this exit status will be success only if all four commands are successful, and failure otherwise. Similar comments can be made for or.

So these can be used for "and"ing or "or"ing a bunch of commands in conditions of an `if` or `while`, or – just as usefully – can be used as a shorthand for simple (possibly nested) `if` statements.

3.1.11 `for in`

The `for` loop iterates a variable over a list of values and runs a body of commands for each one. The syntax is typically

```
$ for VARIABLE in VALUE1 [VALUE2 ...]; do BODY; done
```

3.1.12 Command substitution

This feature is similar to pipes, and is extremely useful and powerful for the same reasons. With command substitution, instead of piping the output of a command into the input of another command, you put the output directly into the line being entered into the shell. This is usually accomplished by surrounding the "sub-command" with backticks (`). For example, if I wanted to give a time stamp to the output file of some command, I could do so with

```
$ some-command > output-`date +%T`.txt
```

where `date` is a command which prints the current date and time according to the given format codes.

3.1.13 Programmable prompt

It can often be nice to have some information in your prompt instead of, for example, a boring old

```
$
```

Modern shells allow you to set the prompt to any string you like, by means of a shell variable. Furthermore, one can put special codes within this string which are expanded into different useful pieces of information, such as – and these are the ones I have included in my shell – user name, host name, and present working directory. See Appendix B for a little more on this.

3.1.14 Miscellaneous features

The rest of the features I've included are too minor for their own sections, so they're all here.

If you want to type some string in your command and be sure that none of it will be expanded or transformed in any other way by the shell, you can surround it by single quotes (`'`). This is especially useful if you have some argument which includes a space; if it isn't surrounded by quotes, it will be treated as two separate arguments.

There is another way to put spaces in an argument, and that is by *escaping* them. This is accomplished by prepending a backslash (\) to the space. This tells the shell to treat the space just as a character, and ignore its special meaning of argument-separator. Other things can be escaped as well. If I put a backslash before a quote, then I'll just get the quote character, rather than starting a quoted string. Newlines can be escaped, allowing a command to be written across two lines. Dollar signs can be escaped so that they don't indicate a variable. Etc. Escaping a normal character has no effect.

Another option for putting spaces in arguments is double quotes (`"`). These are similar to single quotes, except that they are, in a sense, less "strict"; what this means is that certain types of expansions can still occur within double quotes, while others cannot. Notably, variables and command-line substitution can take place, and if the expansions have spaces, the whole thing is taken a single argument, whereas normally it would be space-separated after being expanded.

Lines can be commented out by starting them with a hash symbol (`#`). This isn't very useful when using the shell interactively, but can come in handy when writing scripts.

If an argument is a lone tilde (`~`) or some path (that is, a slash (`/`) separated list of strings) starting with a tilde and a slash, the tilde is expanded to the current user's home directory. This is useful because one spends a lot of time in one's home directory, and is a convenience Unix users are very used to.

That concludes the list of common shell features which I have included in my shell. What follow is a list of certain nice features which I have shamefully omitted, mostly out of laziness. This isn't an exhaustive list of all the features which modern shells have, as such a list would be practically endless. I have only included those that I feel are most important, and that I am therefore most embarrassed about having omitted.

3.2 Missing features

3.2.1 Command line editing

The ability to dynamically modify a line before you enter it is called command line editing, and has become remarkably sophisticated over time. This is largely due to the imitation in modern shells of the two canonical Unix text editors, `vi` and `emacs`. Command line editing has been a pretty major part of the shell

experience ever since it was introduced. It's one of those things that you just can't do without once you have it.

I've failed to include it simply because its is too complicated (or, equivalently, that I'm too lazy). Implementing command-line editing is a lot like building an entire text-editor on top of your shell. Instead of just getting a line of text from standard input, you have to handle each character of input yourself and store it in a buffer which can be modified, and handle all the special key bindings for movement and editing. Besides this, there are many terminal-specific considerations and things of that nature. You need to find out what the control codes are for moving the cursor around in different terminals and make sure that the shell is portable etc.

I felt that tinkering with all that stuff, as interesting as it is, would be too much of a digression from the main project, which is to build a shell.

3.2.2 Completion

This feature is really a part of command line editing, but it is so extraordinarily useful that it deserves a section of its own. Completion is the ability of the shell to complete a word which you are typing, typically by pressing Tab or some other key sequence. The simplest example is just completing the name of a file or a path. But completion can get arbitrarily complex by allowing programmable completion via some kind of scripts. The options to any given command can be arranged to be completed, with all the possible intricacies of the syntax. If, for example, a program expects some option (`-x` or whatever it may be) followed by an `mp3` file, it could be arranged so that only paths to mp3 files will be completed after a `-x`. The script could even access some information on-line and complete, for example, the names of packages to be downloaded from a repository. The possibilities are endless.

Of course, it would be a nuisance if each individual user had to write each of the completion scripts, but of course these things can be compiled and distributed, so that the end-user doesn't have to know anything about the inner-workings of the completion mechanisms.

3.2.3 Command line history

Command line history is the storage of the last however-many commands entered. These can be accessed through key-bindings of the command-line editing interface (often the up- and down-arrows, for instance), or through special sequences of characters. For example, an exclamation mark (`!`) followed by a number typically expands into the line with that index in the history list. Modern history expansion is very sophisticated, allowing you to recall individual words or group of words from any given line, optionally replacing a given string with another one, and all sorts of other neat things.

3.2.4 Globbing

Often, one wants to operate on a whole bunch of files, and it is tedious to name each one of them separately on the command line. For this, special so-called *glob* patterns can be used. In glob patterns, an asterisk (`*`) represents any sequence of characters, a plus sign (`+`) represents any single character, and a question mark (`?`) represents one or zero characters. So if one of the arguments to a command is `a*b?c`, it will be expanded to all the files in the current directory whose name is "a" followed by any (possibly empty) string, then a "b", followed one or zero characters, and then a "c". This is a very hand and widely used feature. I didn't include it because by the time I thought of doing so, the word expansion stuff was already written, and it would have been too complicated to modify it to include globbing.

3.2.5 Job Control

Never mind those other features that I have expressed dismay at omitting. This is the really important one. Job Control is the ability to run more than one process at the same time from the shell. Such a thing appears trivial to users accustomed to graphical environments, with many windows running different programs. In the terminal, however, there is only one place whence to receive input and whither to display output. The solution to this is to have, at any given time, a single so-called *foreground* process, and zero or more *background* processes. The foreground process is the one which receives the input from the shell. By default, when you launch a program, it is the foreground process. But any number of processes can be launched, instead, to the background. Moreover, if a process is already running in the foreground, it can be "stopped" with a special key sequence, at which point execution of that program is suspended and the shell regains control of the terminal (when running programs in my shell, typing such a key sequence has no effect). Thereafter, any stopped processes may be made to continue running in the background, or any one of them can be put in the foreground (typically with the shell built-ins `bg` and `fg`), or even killed.

I have omitted this particularly important feature, again, because of laziness. I would have had to deal with a lot of signal handling, and process and terminal management, which I didn't feel like doing.

3.2.6 Other features

The missing features I've mentioned here serve to paint a fairly good picture of the amazing power of a *real* shell. On the other hand, I'm really just scratching the surface.

There are a number of other features of varying utility that have become standard in modern shells. In addition to that, each shell has its own non-standard features, so that altogether the number of features in a modern shell is truly overwhelming. If you're interested, two of the most popular shells today are *bash* (the bourne-again shell) and *zsh*.

4 jojsh

The original Unix shell was called "sh". It has therefore become a custom to end the names of shells with "sh" (examples include csh, tcsh, ksh, bash, dash, and zsh). Accordingly, given that I go by "Joj", I have called my shell "jojsh".

In this section, I will describe the implementation of my shell. This will include describing the various components of its structure, and showing how they are implemented. This will mean looking at some code. Sometimes I will include the code directly, but sometimes I will just say the name of the function and which file it's in, and in such cases you are encouraged to "read along" while I describe how the code works.

So, without further ado, let us see how a shell works.

4.1 Parsing

4.1.1 Grammars

Before anything else can happen, the first thing the shell must do is parse the input. In order to accomplish this, we must first determine what kind of input we will expect – or put another way, we must define the *grammar* of the shell. A useful and common way of defining such a grammar is to think of the grammar as a certain top-level structure consisting of certain sub-structures, which in turn consist of sub-sub-structures etc. until at last you get to some basic "atomic" elements. In English sentences, for instance, the top-level structure might be *sentence* which might consist of *subject*, *predicate*, and *period*, the former of which might consist of *article*, *adjective*, and *noun*. These last three (along with *period*) would be the aforementioned atomic elements (called "terminal symbols" in technical language).

There is a very convenient language that can be used to express these grammars called *Backus-Naur Form* (or BNF for short). BNF consists of a series of statements which look like

```
SENTENCE = SUBJECT, PREDICATE, "."
```

There is a popular program called *yacc* ("yet another compiler compiler") which takes as input a grammar definition in BNF and generates code to parse such a grammar. But *yacc* expects as input the output from another function called a *lexical analyzer* or *tokenizer*. This is a function which receives the input, and splits it into "atoms", which it sends one by one to the parser (in the examples given above, this would involve maintaining a list of all English adjectives and nouns and deciding which one each word is). There is also a popular tool for generating such lexical analyzers, called *lex*.

Since it was my intention to build this shell "from scratch", I haven't used *yacc* or *lex*, but instead wrote my own parser and lexical analyzer. This wouldn't have prevented me from first defining the shell syntax in a BNF, and using it to write my parser, but as it happens, I didn't do this, but instead "figured it out as I went". That said, there *is* a BNF grammar (in the world of ideas) which

defines the syntax of my shell, and I have (approximately) reproduced it below, in Figure 1. Note that square brackets (`[]`) indicate optional elements, vertical bars (`|`) indicated "or", curly brackets (`{}`) indicate repetition, and parentheses (`()`) are used for grouping; so the definition `A = B, {C} | (D | E), [F]` means `A` consists either of a `B` and one or more `C`s, or of a `D` or `E`, possibly followed by an `F`. I should also note that this isn't *precisely* BNF syntax (nor is what I wrote above), but it comes to the same thing.

```
STMTS = STMT, STMT-TERMINATOR, [ STMTS ]
STMT = CMD-STMT | IF-STMT | WHILE-STMT | FOR-STMT | ASGNMT-STMT
STMT-TERMINATOR = { NEWLINE } | {";"}
CMD-STMT = { WORD }, [ REDIRECT ] [ ("|" | "&&" | "||"), CMD-STMT ]
REDIRECT = [ STRING ], ("<" | ">" | ">>"), WORD
WORD = { PIECE }
PIECE = STRING | DOUBLE | BACK | VAR
DOUBLE = '"', { WORD }, '"'
BACK = "`", STMTS, "`"
VAR = "$", STRING | "${", STRING, "}"
IF-STMT = "if", STMTS, "then", STMTS, [ "else", STMTS ], "fi"
WHILE-STMT = "while", STMTS, "do", STMTS, "done"
FOR-STMT = "for", STRING, "in", { WORD }, STMT-TERMINATOR, "do",
           STMTS, "done"
ASGNMT-STMT = STRING, "=", WORD
```

Figure 1: Retrospective BNF

As I said, this is only approximately the syntax implicit in the program. There are a few minor things that aren't totally accurate (I mention, for example, nothing about white space) but in general, it pretty much sums up the syntax of the shell, and the way it is laid out here is for the most part reflected directly in the implementation of the parser.

There a couple of things to note. One is the recursive definition of `STMTS`. I could have equally used the repetition indicator (`{}`) to say that a `STMTS` is one or more `STMT`s, but I put it like this because it more accurately reflects the actual implementation. Similarly, I could have recursively defined something called `PIECES` instead of having `WORD`, but the way it is done here – with repetition – more accurately reflects the program.

The other thing to note is that there is no definition of `STRING`. This is because it is an "atomic" element, and a definition would simply say that it consists of a bunch of characters. (Strictly speaking, this isn't really true, because `STRING`s can be surrounded by single quotes, which allow them to contain certain extra characters (such as `"` and `$`), so they do have *some* kind of internal structure, but again, this isn't a precise or formal grammar definition.)

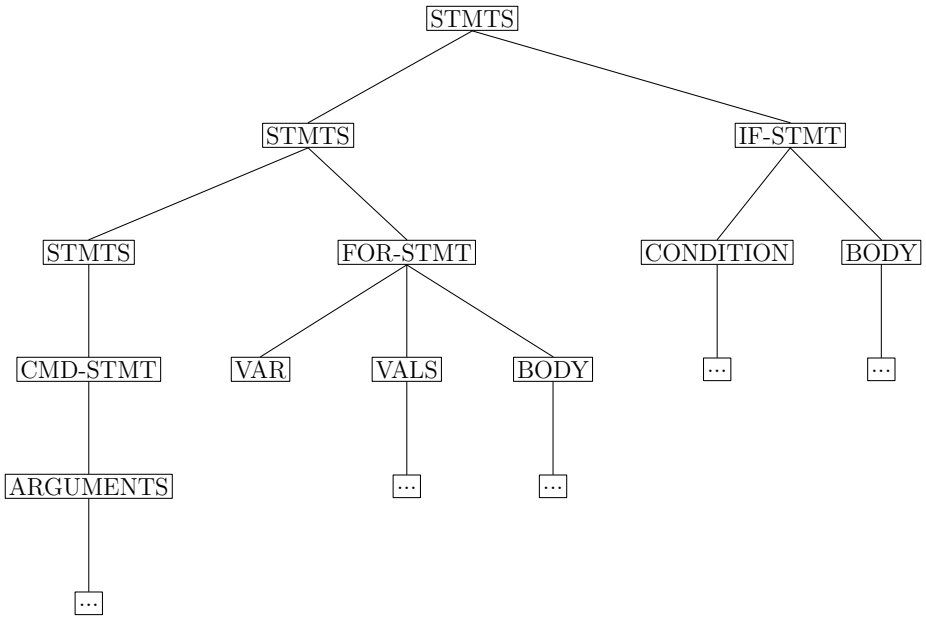

Figure 2: Parse Tree

4.1.2 Parse Trees

Moving on to the actual implementation of the parser. The structure of this kind of grammar lends itself very well to a tree of the sort illustrated in Figure 2.

Because each element of the grammar consists of a number of sub-elements, we can think of each element as a node in this tree, containing all its sub-elements as sub-nodes. The idea, then, is to parse the input and construct such a tree. This would involve defining a `struct` for each type of element, whose members are of the types of its sub-elements. These definitions can be seen in the file `parse.h`.

The one problem with this is that a `STMT`, one of the elements of a `STMTS`, isn't a single type, but can be any one a handful of different types of `STMT`s (see Figure 1). This would suggest an object-oriented approach, with an abstract `Stmt` class, and several subclasses (`IfStmt`, `ForStmt` etc.). The problem is that there is, of course, no such mechanism in C. Fortunately, one can use an ingenious method to "fake" subclasses, which works as follows.

You define a "parent class", which is a `struct` containing a single `enum`, which in turn contains one value for each "subclass". Each "subclass" has as its first member that same `enum`. When an instance of a given "subclass" is constructed, the `enum` field is set to the value corresponding to that type, and then that instance is cast to the parent type. This newly cast object thus now has the type of the parent, but it can still be identified by its `enum` and cast back into its proper type when necessary. In our case, the `enum` is defined like

this:

```
typedef enum {
    ASGNMTS,
    CMDS,
    IFS,
    WHILES,
    FORS,
    EOFS
} StmtType;
```

and the "parent class" is defined like this:

```
typedef struct {
    StmtType stmttype;
} Stmt;
```

The definition of the `IfStmt` "subclass", for example, is then

```
typedef struct {
    StmtType stmttype;
    Stmts *cond;
    Stmts *body;
    Stmts *elsebody;
} IfStmt;
```

(Notice the `StmtType` at the beginning of the `struct`.) A `Stmts` can now simply contain a `stmt`:

```
typedef struct _Stmts {
    Stmt *stmt;
    struct _Stmts *next;
} Stmts;
```

4.1.3 Words and Pieces

Just as there are different types of `Stmt`s, there also different types of `Piece`s. But what is a `Piece`? Well, if we look back at Figure 1, we see that a `Piece` is an element of which there are one or more in a `Word`, and is one of `STRING`, `DOUBLE`, `BACK`, or `VAR`. A `Word`, in turn, is the element of which there are one or more in a `CmdStmt` (and which appears also in other elements). So what do we make of all this?

Well, we said that a command consists of one or more arguments, separated by spaces; these are the `Word`s. But each of these `Word`s can contain certain portions which will need to be expanded and transformed; `Piece`s serve as these portions. A `STRING` is simply a plain string which does not need to be expanded. `VAR`s, `DOUBLE`s, and `BACK`s are, respectively, variables to be expanded, strings surrounded by double quotes, and strings surrounded by back-ticks (for command-substitution). If you look at their definitions in Figure 1, you will

see that they correspond to the nature of these things (which are described in §3.1.8, §3.1.14, and §3.1.12, respectively).

`Piece`s are then defined in the same way as `Stmt`s; the type identifier:

```
typedef enum {
    STRINGP,
    DOUBLEP,
    VARP,
    BACKP
} PieceType;
```

the "parent class":

```
typedef struct {
    PieceType piecetype;
} Piece;
```

and the various "subclasses", for example:

```
typedef struct {
    PieceType piecetype;
    Stmts *stmts;
} BackPiece;
```

I pointed out in §4.1.1 that the definition of `WORD` in Figure 1 used repetition instead of recursion as in the definition of `STMTS`, and said that this was reflected in the program. Well, here's the proof:

```
typedef Piece **Word;
```

That is, `Word` is just defined as a list of `Piece`s. But this leaves the question as to *why* it should be so defined. After all, I could have defined `Word` as follows:

```
/* NOT the definition of Word */
typedef struct _Word {
    Piece *piece;
    struct _Word *next;
} Word
```

which would have been analogous to the definition of `Stmts` shown above. The reason I didn't define it like this is that I felt that it would have been less semantically accurate; whereas the thing following the first `Stmt` in a `Stmts` is another `Stmts` in its own right, the thing following the first `Piece` in a `Word` seemed to me be just a bunch of more `Piece`s and not a `Word`.

There is one final type of node in the tree, which is neither a `Stmt` or a `Piece`. It appears only in `CmdStmt` and appears as `REDIRECT` in Figure 1. In the program, the name of the type is simply `Red`.

Now I have described all of the nodes of our parse tree. From here, it is just a matter of writing some code to build the parse tree out of the input, and then some other code to interpret the parse tree and execute commands.

But before we discuss either of these, we will make a slight digression, and talk about the tokenizer.

4.1.4 The Tokenizer

As I mentioned in §4.1.1, the parser expects its input to come in neat little packages called "tokens", which represent the elements of the grammar. I also mentioned that the input is split up into these tokens by a *lexical analyzer* or *tokenizer*, and that I have written such a thing. I will describe it here. The tokenizer resides in the files `token.c` and `token.h`.

The tokenizer is pretty much a single function, called `getToken()`, whose return type is an `enum` called `TokType` which containing an entry for each token which can be returned. It takes a `char **` as an argument, where it stores a string containing the token itself. For some `TokType`s, such as `WHILE`, `THEN`, `PIPE`, etc. this information is redundant, because the token is a specific string; but in certain cases (most significantly for `WORD`), the string needs to be examined further.

The way the tokenizer works is pretty straightforward and ugly. It simply works through the input, character by character, until it figures out definitively which token is the next one. For several tokens, this can be determined after checking only or two characters (namely, the `TokType`s `SEMCOL`, a semicolon; `PIPE` and `OR`, one or two vertical bars; `NL`, a newline; `RED`, one or two angle brackets (for redirection); and `AND`, two ampersands). If one of these tokens is not matched, the tokenizer continues to get characters until a token-terminating character is met; this is either a space or one of the above mentioned characters. The string thus obtained is then compared to each of the "string-like" tokens (`if`, `done`, etc.), and the appropriate `TokType` is returned. If the string matches none of these, then `WORD` is returned.

This covers most of the functionality of the tokenizer, but there are a couple of other things that it has to deal with. One is quoted strings. The tokenizer has an internal state stored in the variable `QUOTECHAR`, which is defined as follows:

```
enum { NONE = 0,
       DOUBLE = '"',
       SINGLE = '\'',
       BACK = '`'} QUOTECHAR = NONE;
```

As you can see, `QUOTECHAR` is initialized to `NONE`, indicating that the tokenizer is not currently "inside" any quotes. If the tokenizer meets any of the three quote characters while its in the `NONE` state, it changes the value of `QUOTECHAR` appropriately. The effect that this has is that the tokenizer no longer ceases to get characters when it meets a token-ending character, but rather continues until it finds the matching quote character, at which point it sets `QUOTECHAR` back to `NONE` and resumes normal operation.

Another special case that the tokenizer has to deal with is assignments. Assignments, remember, are of the form `variable=value` (§3.1.8). Therefore,

the first thing which the tokenizer checks, when its determining which "string-like" token it has found, is whether the string contains a `=` character. If it does, and all the characters preceding the `=` meet the criteria for allowable variable names, `ASSGNMT` is returned.

Another somewhat complicated special case is redirects. Recall that the angle brackets in a redirect can have a number prepended to them in case you want to indicate from (or to) which file descriptor you want to redirect (§3.1.5). Therefore, if the token-terminating character that stopped the tokenizer was one of the angle-brackets, it adds the bracket to the string, and returns the whole thing as a `RED` instead of a `WORD`.

It should be noted that if the token-terminating character is *not* an angle-bracket, then the tokenizer has taken one character too many, and must "put one back". Fortunately, there is a function `ungetc()` which allows you to do just this – that is, push a character back to be gotten in the next call the `getchar()`. If you look at the code for `getToken()`, however, you will notice that I don't use `ungetc()`, nor do I use `getchar()`. Rather, I use two functions which I defined myself, called `getch()` and `ungetch()`. The reasons for this will be discussed in the next section (§4.1.5).

Two final small things the tokenizer must take into account are comments and escaped characters (§3.1.14). Comments are easy; if the first character of a token is a hash symbol (`#`), the tokenizer gets the rest of the line, ignores it, and returns `NL`. If a hash symbol is met elsewhere, the rest of the line is gotten and ignored, and then the final newline is "ungotten". Escaped characters, remember, are prepended with a backslash (\). So to take care of these, every time the tokenizer meets a backslash, it simply adds the following character to the token string, eliminating the possibility of the tokenizer reacting to any special meaning that character might have.

Just as characters can be pushed back on the input stream with `ungetc()`, this tokenizer allows you to push back tokens in case you took an extra one. This is accomplished with the function `ungetToken()`, which takes as arguments the `TokType` and string corresponding to the token to be "ungotten". It stores these in two static variables called `backtok` and `backstr`. The first thing `getToken()` does, then, is to check if these variables have anything in them, and if so, it just returns their contents instead of getting another string.

I have now described more or less completely how `getToken()` works; but this is not the whole tokenizing story.

4.1.5 Second-pass parsing

The function `getToken()` returns as one of its `TokType`s a token called `WORD`; notice that it doesn't return any `Toktype` called `PIECE`. Since each `WORD` is made of one or more `PIECE`s, the `WORD`s returned by `getToken()` must be tokenized again (into pieces); moreover, `getToken()` isn't able to do it, because it only parses into `WORD`s. There is therefore another function, simply called `getToken2()`, which is very similar to `getToken()`, except it differs in the details of its tokenizing.

This also brings to light the purpose of the functions `getch()` and `ungetch()` mentioned in the previous section (§4.1.4). When you call `getToken2()`, you don't want it to get its input from standard input; rather, you want it to get it out of some string – that is, whatever word you want it to tokenize. The functions in the file `dyninput.c` provide this functionality with a transparent and consistent interface (that is, `getch()` and `ungetch()` work the same way regardless of what's going on underneath). Along with the two previously mentioned functions, there are two other functions in this file called `inputString()` and `restoreInput()`.

`inputString()` takes as arguments the string which you want to set as the source for `getch()`, and the length of that string. It returns a structure called `Source` which is defined in `dyninput.h`. The definition looks like this:

```
typedef struct
{
    int len;
    char *str;
    char ungot;
} Source;
```

Hopefully, this is pretty self explanatory. `str` holds whatever string is associated with this source, and `len` indicates the length. As you get characters, the pointer `str` steps along the string, and `len` is reduced. When `len` gets to 0, `EOF` is returned. `ungot`, of course, stores the character ungotten with `ungetch()`.

The source returned by `inputString()` is the *current* source. Whoever called it is then supposed to back it up somewhere, and when the newly inputted source is finished, the old one is restored with `restoreInput()`, which takes a `Source` as an argument. In the file `dyninput.c`, there is, naturally, a static `Source *` variable called `current`, which points to the current source, and which is initially set to `NULL`. The first caller of `inputString` will therefore get `NULL` as a return value, and will supply the same to `restoreInput()`. Whenever `current` is set to `NULL`, `getch()` simply calls `getchar()`, and `ungetch()` calls `ungetc()`.

So the sequence of actions for second-pass tokenizing is *i*) input the word to be tokenized with `inputString()` (storing the returned `Source` somewhere); *ii*) tokenize it with `getToken2()`; and *iii*) restore the old input source with `restoreInput()`. All that said, the functionality of `getToken2()` is really very similar to `getToken()`. It returns a different set of tokens, and the way it handles quotes is a little different, but its not worth explaining. Look at it yourself if you're interested.

4.1.6 The parser functions

With the tokenizer in place, writing all the parsing functions is fairly straightforward. We won't look at all of them, because some of them are quite similar, but we'll look at a few to get a feel for how they work.

Each parser function has a name like `parseStmts()`, `parseCmd()` etc. and they parse the upcoming input and return a `Stmt` (or `Piece`) object of the type corresponding to whatever they are supposed to parse. Most of the time, one of these functions is called by another `parseX()` function, and the returned node will be entered into the `X` which `parseX()` is parsing. The one exception is the root of the tree, returned to the main loop from the first call to `parseStmts()`, which is then given to `runStmts()` and `cleanStmts()` (§4.3 and §4.4).

These functions are defined in the file `parse.c` and the many data types which they depend on are defined in `parse.h`.

4.1.7 `parseStmts()`

Let us start off by looking at `parseStmts()`. Remember, this is the function which, if we call it at the beginning of the input, should return to us the whole upcoming series of commands in the form of a parse tree. `parseStmts()` takes two arguments, an `int` called `body`, and an `int *` called `err`. The first of these is really just a boolean, indicating whether the `Stmts` to be parsed is inside the body of some construct (like an `if` or `for`). We'll see what this is used for soon. The second is actually also a boolean, and is used for error reporting; if anything goes wrong in the function, `*err` is set to 1, and then the calling function will know something went wrong. We'll see that many functions in the program use this.

The first thing that happens in the function – like in all of the parser functions – is the allocation of the node in question:

```
stmts = malloc(sizeof(Stmts));
```

Now, a `Stmts` node is supposed to contain two things: the first `Stmt`, and another `Stmts` node containing all the rest of the `Stmt`s. To get the former, we'll have to call the parser function for whatever the upcoming `Stmt` type is. We can figure out which type it is just by getting the first token:

```
switch (tok = getToken(&str)) {
```

If, for example, the token is an `IF`, we know we want to run `parseIf()` and so on. If we look inside each `case:`, we see that they almost all have a statement of the form

```
stmts->stmt = (Stmt *) parseIf(err);
```

as we might expect. We see the cast to `(Stmt *)` as explained in §4.1.2; we cast it to an object of the "parent class", and we can cast it back later on using the contained `StmtType`. In the case of `IF`, we also see a

```
free(str);
```

This is because we know what the string of this token is ("if"), and so having no use for the information contained in that string, we free the memory (this implies that `getToken()` leaves it up to the calling function to free the space allocated for the string – which is indeed the case). In other cases, like for `RED` and `WORD`, `str` is not `free`d., but rather we see a

```
ungetToken(tok, &str);
```

This is because the contents of the token *are* useful, so the token is `ungot`ten to be used by the subsequently called parser function.

There are a couple of other special `case`s. One is `SEMCOL` (semicolon). The code for this case is

```
free(str);
free(stmts);
return parseStmts(body, err);
```

This makes sense. A semicolon is never the first token of a `Stmt`; rather it indicates that whatever comes *after* the semicolon is the next `Stmt`. So instead of returning the currently allocated `Stmts` as we normally do, we get rid of it and simply call `parseStmts()` again, which will get the following `Stmt`.

Another special `case` is `NL` (newline). Now, you might expect this to do the exact same thing as with a semicolon – that is, simply move on and get the statement following the newline. But in general, the user expects the typed commands to execute after each line. So when getToken() returns an `NL`, `NULL` is returned, indicating the end of the `Stmts`. If, however, `body` is set to true (non-zero), then all the statements up to the closing token of the body should be executed, and so `NL` is indeed treated as a statement separator just like `SEMCOL`, and similar actions are taken. The difference here is that extra input may be required; if the input is coming from a file, then the next line will be readily available, but if the shell is interactive, the user will have just typed a newline, and must type the following line. The only difference that this makes is that we must present the user with a prompt. This is accomplished as follows:

```
if (INTERACTIVE) {
    fprintf(stderr, "%s", parsePrompt(getvar("PS2")));
}
return parseStmts(body, err);
```

Here we see four things that we haven't seen. The first is `INTERACTIVE`. This is a global variable which is set at the beginning of the program, and which indicates whether the shell is in interactive mode. Next, `getvar()`, as you might guess, is a function which returns the value of the given variable. `PS2` is the name of the variable which stores the special prompt meant for just this purpose (`PS1` stores the normal prompt). Finally, `parsePrompt()` is the function which expands all the special prompt features and returns the final, fully beautified prompt (see §3.1.13).

Similar to `NL` are the cases `THEN`, `ELSE`, `FI`, `DO`, and `DONE`; these indicate the end of a `Stmts`, though while `NL` is for a single-line command, these all indicate the end of bodies. Therefore, if `body` is set to true, these have the same effect as a `NL` has when `body` is false. If `body` is false, then these tokens are unexpected, and it falls through to the `default` case, which gives the user an error message, sets `*err = 1`, and ends the `Stmts` with a `NULL`.

Another special case is `EOFT`. This is a `tokType` so far unmentioned, which `getToken()` returns when it meets an End Of File. Let us look at the code for this case:

```
free(str);
stmts->stmt = (Stmt *) EOFStmt();
stmts->next = NULL;
return stmts;
```

Here, the list of statements is closed and returned, just as with `NL` and the body-enders, except that the last `Stmt` is made an "`EOFStmt`". An `EOFStmt` is a node with nothing it except the `StmtType` identifier, which contains the value `EOFS`. Its purpose is simply to indicate that the input is completed after the previous statement; that is, if one of these is reached while running statements, the shell exits.

After the appropriate statement type has been determined and `stmts->stmt` has been filled, the function ends with

```
stmts->next = parseStmts(body, err);
return stmts;
```

which I hope is self-explanatory.

4.1.8 `parseCmd()`

I hadn't realized how tedious it would be to describe one of these functions, so I'll try to do as few as possible. It should be easier now that we've seen the basic elements of the parser function.

`parseCmd()` is particularly important, because as we both know, the shell is really all about running commands. The contents of a `CmdStmt` are as follows:

```
typedef struct _CmdStmt {
    StmtType stmttype;
    Word *args;
    struct _CmdStmt *next;
    enum {PIPEN, ANDN, ORN} nexttype;
    Red **reds;
} CmdStmt;
```

Pretty serious node! `next` is the command, if any, to be piped into or to be executed conditionally on this command's success or failure. `nexttype` says which of these three, if any, is the case. The last entry is the list of all the redirections to take place on this command (since there can be any number).

The first thing that happens in `parseCmd()`, just like in `parseStmts()`, is the allocation of the node. The next thing that happens, however, is very important and characteristic of all the parsing functions *except* `parseStmts()` (and `parseWord()`): the `stmttype` of the newly allocated `stmt` is set (in this case, to `CMDS`). This is so important because, after this `CmdStmt` is casted to a `Stmt`, this will be the only thing making it a `CmdStmt` at all.

Next, the function goes through `getToken()`'s until it gets to a `SEMCOL`, `NL`, `PIPE`, `AND`, or `OR` (any of the things that mark the end of a command). Each token thus obtained is then `switch`ed. If it is a `WORD`, this very exciting line is executed:

```
cmdstmt->args[argc++] = parseWord(err, str);
```

That is, the next argument in this command is set to the obtained string, after being parsed into a `Word` (which, as we remember, consists of one or more `Piece`s, §4.1.3). We'll look at `parseWord()` shortly (§4.1.9). The other case is if the token is a `RED`, in which case we have the very similar

```
ungetToken(RED, &str);
cmdstmt->reds[redc++] = parseRed(err);
```

I will *not* go into the details of `parseRed()`, though it is kind of interesting. Suffice to say that it figures out what kind of redirection it is, and from which file descriptor and to which file or file descriptor it's happening (§3.1.5), all of which information it encapsulates in a nice `Red` structure which it returns.

Note that `parseWord()` takes a string as an argument, rather than having the thing `ungot`ten only to be gotten again. This is the case with those parsing functions where all the parsing is to be done on a single token.

After that, `parseCmd()` checks to see if the next token is a pipe or a `&&` or `||`, and if so sets `nexttype` appropriately and calls `parseCmd()` again (storing the returned `cmdStmt` in `next`).

Finally, it ends with the usual

```
return cmdstmt;
```

4.1.9 `parseWord()`

`parseWord()` is quite like `parseStmts()`, in that its only job is to figure out which kinds of things need to be parsed, and to call the appropriate functions. A significant difference, however, is that all the `Piece`s are contained in a *single* word (see §4.1.3), so that `parseWord()` doesn't call itself recursively to get each `Piece`, but rather loops through and builds up the list of `Piece`s all at once.

Another important difference is this pair of lines which appear at the beginning and end of the function:

```
bu = inputString(s, strlen(s));
/* ... */
restoreInput(bu);
```

This is the switching of input described in §4.1.4. `bu` is, of course, declared as a `Source *`.

Beyond this, the functionality is pretty much identical to `parseStmts()`; for every token `parseWord()` receives, it checks which kind of token it is, calls the appropriate parsing function, and stores the returned `Piece` as the next piece in the `Word` which is being built.

The function `parseDouble()` which parses strings enclosed in double quotes (see §3.1.14) is almost identical to `parseWords()`. After all, a string in double quotes contains strings that may need to be expanded, so it is in reality just a series of `Pieces`; the only difference is that certain characters – in the case of my shell, only spaces and tildes – are not treated specially. Also, things that *are* expanded should not be space-separated into different arguments, but this doesn't require any work from the parser – rather, the simple fact that it *is* a double piece, and not a word, causes the splitting to be disabled when expansion time comes (§4.3.1).

There's one other piece-parsing function I would like to look at, because it's particularly interesting.

4.1.10 `parseBack()`

This function parses a string surrounded by back-ticks, which, as you remember, is replaced by the output of whatever command it contains. (§3.1.12). This is a fairly short function. The meat of it is in these lines

```
/* need to get rid of surrounding quotes */
bu = inputString(strdup(str + 1), strlen(str) - 2);
backpiece->stmts = parseStmts(1, err);
restoreInput(bu);
```

Here, we see another use of `inputString()` besides parsing `Words`. The comment refers simply to the `+1` and `-2` in the code, which serve to remove the back-ticks appearing at the beginning and end of the string.

Once we've removed these, what's left is a string consisting of one or more commands; or, in other words, a `Stmts`. So we simply set that string as input using `inputString()` and then parse it with `parseStmts()`. That's it!

Nice!

The other parsing functions work pretty much according to the same principles as the ones I have shown already. Each has its little quirks, but you can basically guess what each does from the fields contained in the corresponding node type.

So now we've parsed our input! We've built our parse tree, and all we've got to do is run it! Before we get to that, however, I would like to make a small digression.

4.2 Variables

In this section, I will describe how shell variables are stored and retrieved in the program. The functionality described here will be used in the running functions in the next section, so it is good to see how it works here first. All the variable stuff can be found in the file `variables.c` (there's also a file called `variables.h`, but this just contains a few function prototypes to be included in other files).

The two most important functions in `variables.c` are `getvar()` and `setvar()`. The first of these we have already seen in §4.1.7, where it was used to retrieve the value of the variable `PS2`. Both of these functions make use of a hash table (see Appendix C). They are quite simple. `setvar()`, which takes two string called `var` and `val` as arguments, checks if the entry `var` is already in the table, then sets it to `val` if it is there or makes a new entry if it isn't. `getvar()` takes one string, `var`, and returns the value contained in the entry `var` if it exists, and returns an empty string otherwise.

But there are a couple of other interesting functions in this file. One is called `loadenv()`, and it looks like this:

```
void loadenv()
{
    char **env = environ - 1;
    char *var, *val;
    while (*++env) {
        var = *env;
        val = *env;
        while (*++val != '=');
        *val = '\0';
        setvar(var, val+1);
        *val = '=';
    }
}
```

The purpose of this function is to store all currently set environment variables as shell variables (see §3.1.8). This function is called just once at the beginning of the program. The variable `environ` is a global `char **` variable declared in the file `unistd.h`. The C run-time environment sets this variable to the list of all the variables in the environment and their values (in the form `VAR=VAL`). The function is pretty self-explanatory.

The last function in this file is `parrsePrompt()`, mentioned in §4.1.7, which parses the special programmable prompt strings and returns the expanded prompt. This isn't all that interesting; it just goes through, expanding the special codes when necessary with `getlogin()`, `gethostname()`, and `getcwd()`, all of which are standard library functions declared in `unistd.h`. See Appendix B for a little more on this.

4.3 Running

Now that we've got that over with, we can move on to the most exciting part of the shell – actually running the commands! All this stuff is in `run.c`.

For each of the `parseStmt` functions, there's a corresponding `runStmt`, and for each of the `parsePiece` functions, there's a corresponding `expandPiece`.

The initiator of the whole process is, unsurprisingly, the function `runStmts`, which takes a `Stmts` as an argument. It is exceedingly simple. All it does is

check the `StmtType` of the `Stmt` which is in the supplied `Stmts`, give that `Stmt` to the appropriate `runStmt` function, and finally return with

```
return runStmts(stmts->next);
```

Just like `parseStmts()`, then, it calls itself recursively until it has gone through all the statements. The return type of this function is an `int`. Since it returns the result of a recursive call to itself, the value is of course determined by whichever `runStmts()` receives the end of the list. When this happens, it can return two possible values. If the list ends with a `NULL` entry, 0 is returned. If the list ends with an `EOF` statement (see §4.1.7), it returns 1. This way, whoever calls `runStmts()` can know whether an End Of File has been reached (in which case the shell should exit), or whether there's still more statements to be parsed and run.

Of course, the most interesting function that `runStmts()` calls is `runCmd()`. We will see how this function works in a moment, but first we will look at its first line:

```
argv = expandWords(cmd->args, 1);
```

4.3.1 `expandWords()`

The first thing that `runCmd()` does is call `expandWords()`. It should be noted that the variable `argv` above is a `char **`. `expandWords()`, then, takes a list of `Word`s, expands all the `Piece`s, and then returns the expanded list as normal strings. The second argument simply indicates whether word-splitting should take place (its set to zero when `expandWords()` is called by `expandDouble()`).

This function is to `parseWord()` what `runStmts()` is to `parseStmts()`; it is the function which calls all the other `expandPiece` functions.

The way this function works is as follows.

First, it allocates a single buffer where it will store all the expanded words. It then loops through each `Word` in the list supplied to it. For each word, the index of the current position in the buffer – that is, the place where the word will be stored – is added to an array of integers called `expandedidxs`. Then, it loops through each `Piece` in the current word. Based on what's in the `piecetype` field, it calls the appropriate `expandPiece` function (each of which returns a string), and appends the returned string onto the big buffer. Then, space-separation takes place if necessary (that is, if the piece was a back-tick quoted string or a variable, but not if it was a plain string or a double-quoted string). Space-separation works as follows.

It goes through the buffer character by character, starting at the beginning of the piece which is being split, and if it sees a space, it does two things. First, it replaces the space by a null byte (ie. a string terminator). Then, it adds the index of the following character to `expandedidxs`.

This process continues through all the `Piece`s and all the `Word`s, until finally, it ends up with a big buffer with all the expanded and separated strings, along with a list of indices into that buffer where each separated string starts. Then,

an array of `char *`'s, called `expanded`, is built up by taking a pointer to each position in the big buffer which is referred to in `expandedidxs`. It is finally this list of strings – that is, `expanded` – which is returned.

There are a couple of questions to be asked here. The first is, why copy all the strings into this buffer? Why not just build `expanded` directly out of the strings returned by the `expandPiece` functions? This is not hard to see: the reason is that we can't know beforehand how many arguments will be contained in a given `Piece` or, alternatively, how many `Piece`s will be in a given argument. The only way to resolve this is to concatenate them all, and then separate them afterwards.

The next question is, given that we are using this big buffer approach, what's with this strange process of first building a list of indices, and then finding the corresponding pointers afterwards? Why not just gather the pointers for `expanded` at each juncture where indices are now being gathered for `expandedidxs`? This is a subtle point. The reason is that the buffer is continually being reallocated in order to make it bigger and fit more strings. When `realloc()` is called to do this, there's no guarantee that the buffer will be in the same place. So if pointers to within the buffer are taken during the process, there's a good chance those pointers won't point anywhere useful by the time it's over. The present system only gets the pointers *after* the buffer has been reallocated for the last time.

I only want to look at one of the `expandPiece` functions, because it is quite neat.

4.3.2 `expandBack()`

You'll recall that the contents of back-tick quoted strings are supposed to be treated as commands to be run and replaced by their output (§3.1.12). A `backPiece()` in fact, just contains a single `Stmts`. What `expandBack()` does, then, is as follows. First it opens a new pipe. It then closes standard output, and sets it to the write-end of the pipe (via `dup(pipefd[1])`, see §2.6). Then it calls `runStms()` with `back->stmts`, which will run the commands and write all the output (unwittingly) into the pipe. Then, it just calls `read()` on the read-end of the pipe, storing the results in a buffer, until it reaches an end of file, at which point it restores the old standard input, and returns the buffer. That's it!

Nice!

With `expandWords()` in place, `runCmd()` can expand the `Word`s contained in the `CmdStmt`, and get a regular old list of arguments in the form of strings. Let's see what it does next.

4.3.3 `runCmd()`

Let us quickly recall what this function is supposed to do; well, obviously, it runs whatever command is contained in the given `CmdStmt`, but what does that entail?

The command may be a built-in command, in which case whatever actions are supposed to be performed should be performed. Otherwise, the command is an executable to be run. Additionally, all indicated redirections must be performed, and the output possibly must be piped into *another* command. Also, upon completion there's a chance that another command will have to be run (because of a `&&` or `||`). So, let's see how this is done!

Recall that we've already looked at the first line, which expands all the words. Now we have a nice, simple, `NULL` terminated list of strings representing the arguments to the command. The first thing that happens is that the first argument is checked to see if it is one of the built-in commands, which include `cd`, `exit`, `export`, and `source` (or "`.`").

If it's `cd`, then the second argument is supplied to the standard library function `chdir()`.

If it's `exit`, the shell simply exits with `exit(0)`.

If it's `export`, we recall that this indicates that the following arguments are variables that should be exported to the environment. This is done simply by retrieving the value of the variable with `getvar()` and then calling `setenv()` (another standard library function), which takes two arguments (variable and value).

The last built-in command is `source`. This command says that the supplied file should be read, and all the commands therein should be run. This pretty much means running `jojsh` all over again; indeed, the code for this command is almost identical to the main loop of the program. But since I'm saving the main loop for the end, I won't say any more about this.

Now we move on to running executables. The way this is *basically* done – using `fork()` and `exec()` – was already outlined in §2.5.

To get an idea of how it works when multiple pipes are involved, refer to Figure 3. In this diagram, each large box represents a process. All the boxes on the left (containing "jojsh") are the same process – although those connected by solid arrows are in different stack frames – whereas each box on the right is a separate process. The small white rectangles represent the terminal, while matching colored rectangles represent both sides of the same pipe. The rectangles which are on top and underneath a given box are standard input and standard output, respectively.

What is depicted is the running of a pipeline including four commands.

The process starts with the box on the top left(1), which is `jojsh` running `runCmd()`. The first thing it does is open the red pipe. It then `fork`s, producing the process on the top right (i), which **inherits its pipes**, as is always the case with forked processes. This process closes one end of the pipe, and sets the other end as standard output. Then, it calls `exec()` with the list of strings `args` which was returned from `expandWords()`.

In the meanwhile, `jojsh` (A), closes one end of the red pipe, and sets the other end as standard input. Then, it calls call `runCmd()` with `cmd->next` – that is, the next command in the pipeline – as an argument. This next instance of `runCmd()` (2) opens the blue pipe. It then `fork`s, creating the process to the right of it. Not only does this process (ii) inherit the blue pipe, it also inherits

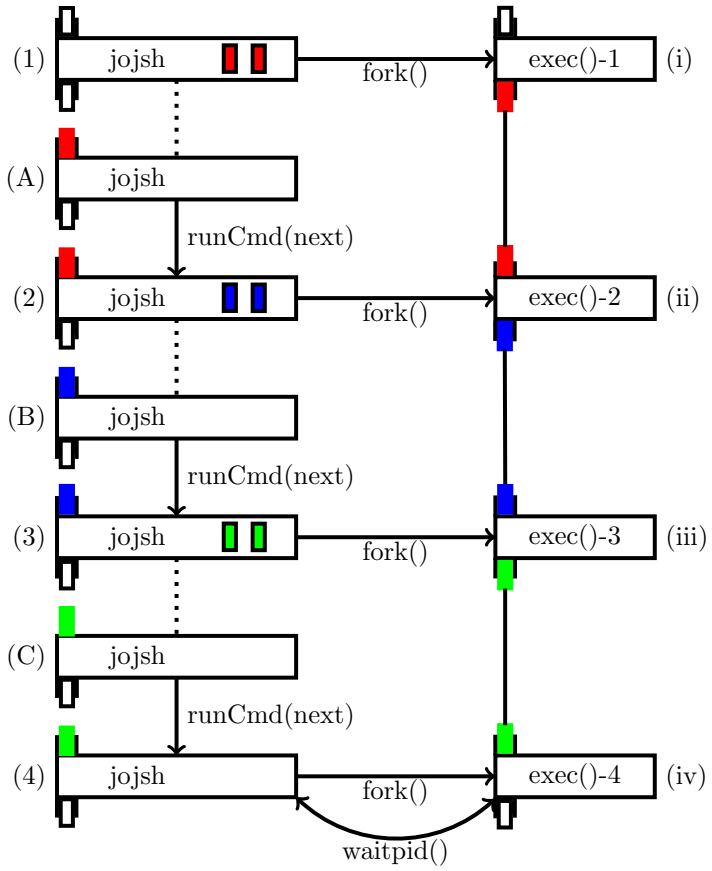

Figure 3: Running programs with pipes

standard input and standard output – as is always the case with forked processes – which are the red pipe and the terminal, respectively. It then closes one end of the blue pipe blue pipe, and sets the other end as its standard output. Then it calls `exec()` again, with the next program. Thus, the processes (i) and (ii) are connected by the red pipe, which is the effect we wanted.

In the meanwhile, `jojsh` (2) closes one end of the blue pipe, sets the other end as standard input (B), and then calls `runCmd()` again with the next command in the pipeline; and so the process repeats again.

The fourth instance of `runCmd()` (4) sees that `cmd->next` is empty – that is, that this `CmdStmt` is the last one in the pipeline – so it does something different. First of all, it doesn't open any new pipes. Then, after it forks, the forked process (iv) *also* sees that there's no `next` command, and so it doesn't change

its standard output, but leaves it rather as the terminal. Then it calls `exec()` as usual.

Meanwhile, `jojsh` (4) – since it doesn't have any `next` command to run, executes the system call `waitpid()` with the process ID of (iv) as an argument. This results in `jojsh` hanging until (iv) exits, at which point the exit status – among other information, which is discarded – is returned, which `runCmd()` saves in the variable `?` with `setvar()`. Then (and this isn't indicated in the diagram), `runCmd()` (still (4)) calls `wait()` – another system call – repeatedly, which causes it to hang until all the other child processes are also finished. Then, (4) returns.

The next thing that happens also isn't shown in the figure: the previous instance of `runCmd()` (C), after the `runCmd()` which it called (4) returns, changes the standard input back to what it was before – the blue pipe. Now, this doesn't actually have any positive effect, but *every* instance of `runCmd()` does this, so (B) changes the input back to the red pipe, and then – and this *does* have a positive effect – (A) changes it back to the terminal. The positive effect in question here is restoring the input back to the terminal, which is where we want it to be.

This is the essence of how running programs works. Of course, if there is no pipeline at all, but rather just a single program, this is equivalent to a reduced version of Figure 3, with only boxes (4) and (iv). There are some small details in `runCmd()` involving process groups, but I haven't mentioned anything about those, nor do I want to, so I won't say anything more about that.

I *will*, however, say something about a couple of other details of `runCmd()` which are to do with redirecting and the `&&` and `||` operators.

The way redirection works is pretty straightforward: before each child process (the roman numerals in Figure 3) calls `exec()`, it checks to see if there is anything in `cmd->reds` (which you'll remember is of the type `Red **`, §4.1.8), and if there is, loops through each `Red` therein. For each one, it calls `close()` on the file descriptor which is being redirected, and then calls `open()` on the specified file, or `dup()` on the specified file descriptor (see §3.1.5). After it has done this for each `Red` in `reds`, it goes on to execute the program.

Just before the end of `runCmd()` – that is, after everything that happens in Figure 3 – there is by this time a value store in the variable `?` – namely, the exit status of the last program in the pipeline ((iv) in the diagram). Depending on the value of this variable, and on whether `cmd->nexttype` is `ANDN` or `ORN` (or neither), `runCmd()` may or may not call `runCmd()` again, with `cmd->next` as an argument.

That's it! We can now run commands, which is what we came here for. The other `runStmt()` functions aren't all that interesting. Briefly, `runIf()` and `runWhile()` use `getvar("?")` to decide whether they should call `runStmts()` on the contents of their `body`s, `runFor()` calls `setvar()` and `runStmts()` over and over, and `runAsgnmt()` pretty much just calls `setvar()`.

4.4 Cleaning

After that exciting thrill ride, we now move on to what is by far the most boring part of the program – cleaning up. What cleaning up, you ask? Well, as I mentioned in §4.1.7, `getToken()` leaves it up to the caller to free the memory allocated for the returned string. Therefore, there are a bunch of pointers to `malloc()`'d memory dispersed all around the parse tree, and this needs to be freed. Besides this, there are the nodes themselves, which have been allocated and are equally spread around the tree.

To free all this memory, then, we just descend into the tree in the same manner as we did for running and parsing. That is, for every `parseX` and `runX` (or `expandX`), there is also a `cleanX`, and they again all call each other recursively like crazy. These functions are contained in the file `clean.c`. I will not say much about these functions, because they are pretty obvious. All they do is call the appropriate other functions (`cleanStmts()` for example, will call any one of the `cleanStmt` functions, depending on the value of `stmttype`), free any `char *`'s they might have, and then free whichever node they've been given.

4.5 The main loop

Now we finally get to see it all come together.

The `main()` function is located in the file `jojsh.c`. The first thing it does is check to see if there are any command line arguments. If there are, it closes standard input, and tries to open the file indicated by the first argument (§3.1.7). If this is successful, then `argv` is incremented by 1, and `argc` is decremented by 1. This, of course, shifts the command line arguments by one, and the purpose is to ensure that references to positional parameters (§3.1.8) in the given shell script will point to the correct arguments.

Next, we set the external variable `INTERACTIVE`, mentioned in §4.1.7 and used throughout the program, which indicates whether the user is interacting with the program, or a script is being read. It is set like this:

```
if (isatty(0))
    INTERACTIVE = 1;
```

`isatty()` is a standard library function which tells you whether the given file (standard input, in this case) is a terminal. `INTERACTIVE` is by default set to 0, so it will only be set to 1 if the input is coming from a terminal, which is what we want.

Then the environment variables are loaded with `loadenv()` (§4.2), and any special parameters (§3.1.8) are set, including `$` (using the system call `getpid()`), the positional parameters, and the default prompt strings in `PS1` and `PS2`.

Then we begin the main loop. I'll reproduce it here in full, because it is fairly short.

```
while (1) {
    if (INTERACTIVE) {
```

```
            fprintf(stderr, "%s", parsePrompt(getvar("PS1")));
        }
        err = 0;
        stmts = parseStmts(0, &err);
        if (! err) {
            if (runStmts(stmts))
                break;
        } else {
            char *str;
            /*in case of error, chomp rest of input,
              or give up if we're not interactive */
            if (INTERACTIVE) {
                while (getToken(&str) != NL)
                    free(str);
            } else {
                break;
            }
        }
        cleanStmts(stmts);
    }
```

This is pretty straightforward. I'll leave it as an exercise to the reader to figure out what's going on. (Just remember that `runStmts()` returns 1 if it ends in an EOF and 0 otherwise, §4.3).

5 Concluding remarks

I have now written a Unix shell. If I wish, I could replace the shell I use normally (`bash`) with `jojsh`, and use it for all my shelling needs. Of course, I won't do that (yet), because I would feel utterly crippled by the lack of features, but the point is that I could.

And this fact demonstrates a very important and beautiful aspect of the Unix operating system: its transparency. The shell is a significant part of the Unix experience, and it can quite easily be implemented and replaced, because it is totally clear and known exactly what it does.

We could continue to do this, implementing and replacing little bits of Unix at a time, until nothing is left but what we've written ourselves (admittedly, certain parts – the kernel, for example – would be more difficult than others, but even this has been done, Linux being an example of just that).

Its likely, in fact, that this very openness is what has fuelled the development and gradual improvement of Unix shells toward the greatness they have today.

A Device Files

In §2.3 and §2.4, I make reference to special "device files" which communicate with devices when read from or written to.

I explained the strange concepts of "standard output" and "standard input" as simply being the device file representing the terminal. But what is this file, and how does it manage to do anything?

Well, each device file has associated it with it two numbers called the *major number* and the *minor number*. The major number indicates what kind of device it is. To make a new device driver on the system, I register one of these major numbers. This consists of writing a bunch of functions which correspond to the different file operations. That is, I write a function for what to do when the device is opened, read from, written to, etc. The content of these functions is all the nitty-gritty low-level device stuff, like writing to the appropriate memory locations which are mapped to the ports of interest.

When I write to a device file, then, the system just checks what the major number is, and calls the corresponding function. The minor number is just a number which is passed to these functions. These can be used for a variety of purposes. A common example is if there's more than one instance of a particular device, the minor number will indicate which device it is (which means there will be one device file for each of these devices, since every device file has its own minor number).

We can see that these device files can not only write and read from hardware, but do just about anything, since I'm allowed to assign pretty much arbitrary functions for the various file operations associated with a major number (in fact, they're not really called "device" files at all, but rather *char* or *block* files). An example of this, in fact, is the one I gave in §2.4: virtual terminals. There, I very mysteriously said that a virtual terminal device file just sends and receives data to and from another process. Well, I still won't go into the details of how this works (though it is quite neat), but hopefully this isn't so hard to believe now, given that any old code at all can be run when writing or reading to and from a device file.

B Programmable prompt codes

In §3.1.13, I mentioned that there are certain special prompt codes which can be put in the prompt string to be expanded to some useful information. The nature of these codes doesn't seem to be standardized or consistent from shell to shell, so I didn't want to mention it in the section about programmable prompts and give the false impression that the specific codes were as standard as the feature itself is. I've therefore included them here, so that they wouldn't be undocumented. In my shell, I only have 4 codes, and they correspond to the ones used in `bash`, which is the default shell in Linux, and the one which I use.

Note that for the last two, if the current directory is the home directory (or a sub-directory thereof, in the second case), it is abbreviated with a tilde (`~`).

Table 1: Programmable prompt codes

Code	Meaning
\u	Username of current user
\h	The hostname of the machine
\W	The basename current working directory
\w	The full path to the current working directory

I also failed to mention in §3.1.13 the names of the shell variables which store the prompt strings (though these are mentioned elsewhere in the text). The main prompt is stored in the variable `PS1`. A secondary prompt string, for multi-line input (like in the body of loops), is stored in `PS2`. I presume `PS` stands for prompt string. I just chose these names because they are quite common across shells.

C Hash tables

It says in the introduction that I presume a knowledge of basic programming concepts, and hash tables are pretty basic, and moreover aren't a shell-specific concept, so I didn't include this in the main body of the text. On the other hand, they aren't *as* basic as the other things I presumed knowledge of, so I figured I'd provide a discussion of them just for completeness.

Hash tables are an efficient (and ingenious) way to store arbitrary amounts of data and look it up quickly. A hash table consists of a series of *linked lists*. A linked list is a type of list in which each element of the list contains some data and a pointer to the next element. This allows you to step along the list in one direction. In a hash table, one of the data in each node of the list must be a name (for looking-up purposes).

The idea with a hash table is that the name of any given entry can be transformed into a single number, in a random but consistent manner; that is, it's very hard to guess what number a given string will give you, but if you transform the same string twice, you'll get the same number. This transformation is called *hashing*.

To store a given entry in the hash table, the name is first hashed, and the returned number indicates the index of a linked-list in the hash table. The list is stepped-along until a free entry is found, and then the entry is stored there.

When it comes time to look up a given entry, it is hashed again, and the list with the index corresponding to the returned number is searched. If the entry in question isn't found in the list, then it is certain that that entry can't be anywhere else in the table, because it *has to be in the list corresponding to its hash value*.

We might ask what the point of this is. Why not, for example, just use a single linked list? The disadvantage to this is, of course, that one would have to go through *all* the entries in order to get to an entry which is quite late in

the list. Another possibility is to still have multiple lists, but index them in a more natural way; for example, each list represents the first letter of the entries it contains, and to look up an entry, I just look at its first letter and go to the corresponding list. The problem with this is that there would be an uneven distribution; while it would be very quick to look up entries starting with "q", it would take a very long time to look up entries starting with "t", because that would be a very big list.

With hashing, the distribution is random, so the lists are guaranteed to be pretty similar in size. Furthermore, they can be made arbitrarily small on average by simply having more lists, thus splitting up the entries more.

I must confess that the hashing algorithm I used is taken almost directly from the book *The C Programming Language* by Brian Kernighan and Dennis Ritchie (the only C book worth reading). It resides in the function `hash()` in `variables.c`, and looks like this:

```
for (hashval = 0; *s != '\0'; s++)
    hashval = *s + 31 * hashval;
return hashval % 101;
```

What this algorithm essentially does is to treat the given string as a number in base 31, each letter being treated as a digit corresponding to its place in the alphabet, and then taking the remainder of the whole thing divided by 101. The specific numbers don't necessarily have to be 31 and 101, as long as they are relatively prime (share no common factors) – a condition which is assured if they are both prime, which is the case here. The reason for this is to get all possible values for the modulo. (My choice for these specific numbers is due to the fact that they were the numbers used in the source from which I plagiarized this).

It is clear in this algorithm that the resulting number will have very little to do with the initial string in any meaningful way, which is exactly what we want. The number of lists in the hash table will of course be in this case 101.